GoAhead: A Partial Reconfiguration Framework

Christian Beckhoff, Dirk Koch and Jim Torresen
Department of Informatics, University of Oslo, Norway
Email: {christian}@recobus.de, {koch,jimtoer}@ifi.uio.no

***Abstract*—Exploiting the benefits of partial run-time reconfiguration requires efficient tools. In this paper, we introduce the tool GoAhead that is able to implement run-time reconfigurable systems for all recent Xilinx FPGAs. This includes in particular support for low cost and low power Spartan-6 FPGAs. GoAhead assists during floorplanning and automates the constraint generation. It interacts with the Xilinx vendor tools and triggers the physical implementation phases all the way down to the final configuration bitstreams. GoAhead enables the building of flexible systems for integrating many reconfigurable modules very efficiently into a system. The tool targets (re)usability, portability to future devices, and migration paths among reconfigurable systems featuring different FPGAs or even FPGA families. Moreover, it provides a scripting interface and all features can be accessed remotely.**

I. Introduction

The overall goal of partial reconfiguration (PR) is to share FPGA resources among multiple modules that are executed mutually exclusively to each other. By time-sharing FPGA resources, more functionality can be put into a single FPGA or, alternatively, a given functionality can be implemented on a smaller device. A higher level of integration or a smaller device results in a more cost effective implementation. Moreover, this helps to save power. In the latest CMOS process technology, in which static power consumption dominates overall power consumption, the power savings are roughly proportional to the area savings and even the power overhead for the reconfiguration will become more negligible with future CMOS processes where the static to dynamic power ratio will probably continue to rise [1]. Therefore, partial reconfiguration is in particular of interest for many embedded systems that have strict power or cooling requirements.

There are several more use cases for partial reconfiguration; however, the real challenges arise with the implementation of corresponding systems. First of all, a run-time reconfigurable system must improve a static only solution. This requires that the logic overhead for the communication with the reconfigurable modules is kept low. Besides the efficiency of the reconfigurable system, it is similarly important to have an efficient way to implement reconfigurable systems. This demands tools and design flows that are able to implement efficient design methods for reconfigurable systems. For more than a decade, a lack of tools has hindered the wide use of partial reconfiguration and this technique was almost completely restricted to academic projects where researchers with much low level expertise created remarkable systems (e.g., [2], [3], [4], [5]).

However, the two FPGA vendors Xilinx and Altera are now providing reliable tools that hide all low level FPGA details from the design engineer [6], [7]. In short, both flows are based on an incremental design methodology where the static system (the part of the system that will be present at any time, e.g, a CPU or memory controller) will be implemented in a first step. In order to implement the communication to and from the reconfigurable modules, an anchor LUT (called *proxy logic* by Xilinx) will be added into the reconfigurable regions and connected, as shown in Figure 1. The partial modules are then implemented as an increment to the static systems (where the placement and routing will be preserved).

This is nicely supported by the tools but has three major restrictions. 1) As the routing to the anchor LUTs is not strictly constrained, it is in general different for each reconfigurable area, as sketched in Figure 1. Furthermore, there might be static signals crossing a reconfigurable area. This prevents module relocation among different reconfigurable areas, even if the different reconfigurable modules would be identical for both areas. 2) The routing to the anchor LUT will in general change each time the static system is changed. Consequently, all permutations of module instance and placement position have to be reimplemented on each change of the static system. Finally, 3) a reconfigurable area can only host one module exclusively (island style reconfiguration) and it is not supported to share a reconfigurable area by multiple modules in a flexible manner at the same time.

These restrictions have been circumvented by a few academic tools (e.g., ReCoBus-Builder [10] or OpenPR [9]) that use hard macros for the interface routing and special *blocker macros* to constrain the routing. With these tools, module relocation, multi module instantiations, and sharing a reconfigurable area by multiple modules are possible. However, these approaches ignore some important design aspects like the internal module timing and there are new challenges with recent Xilinx FPGAs. Table I lists the most important features of the old and new Xilinx design tools together with the ReCoBus-Builder and OpenPR. The last column includes the features provided by our new tool GoAhead , which will be introduced throughout the next sections. GoAhead provides important features for better utilizing partial run-time reconfiguration and it supports the latest devices, including

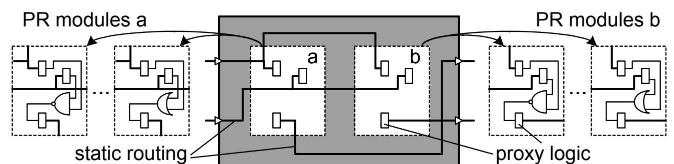

Figure 1. Partial module integration using proxy logic [6].

Table I
COMPARISON OF DIFFERENT INDUSTRY AND ACADEMIC PR DESIGN TOOLS WITH OUR NEW TOOL GOAHEAD

feature	PlanAhead (old) [8]	PlanAhead [6]	OpenPR [9]	Recobus-Builder [10]	GOAHEAD
supported devices	V-II/Pro V4	V4, V5, V6	V4, V5	V-II/Pro, S3	V4[3], V5, V6, V7[3], S6
floorplanning GUI	yes	yes	uses PlanAhead	yes	yes
script interface		TCL			TCL-like
module relocation	no[1]	no	yes	yes	yes
static/partial decoupling	no[1]	no	yes	yes	yes
partial region crossing	yes	yes	no	no	yes
hierachical reconfiguration	no	no	no	no	yes
component-based design	no	no	no	yes	yes
communication method	bus macro	proxy logic	bus macro	various macros	proxy logic, bus macro, direct wire
communication architecture	no	no	no	bus/streaming	streaming
reconfiguration style					
single island	yes	yes	yes	yes	yes
multi island	no[1]	no	yes	yes	yes
slot-based	no[1]	no	yes[2]	yes	yes
grid-based	no[1]	no	no	yes	yes

[1]: Module relocation and advanced reconfiguration styles are natively not supported, but has been shown in [5], [11];
[2]: Not well supported due to a missing on-FPGA communication architecture.
[3]: Due to lack in available FPGA boards, these devices are only verified using the Xilinx design tools, but have not been tested on an FPGA.

Xilinx Spartan-6 FPGAs that are not even supported by the Xilinx vendor PR flow. The tool, documentation, and example systems are available on our project website [12].

In the next section, we will introduce the tool and the GOAHEAD design philosophy. After this, some distinguishing features will be presented in dedicated sections, including a methodology to avoid bus macros or proxy logic (Section III), the technique to relocate modules in GOAHEAD (Section IV), and an approach to permit static signals to cross areas hosting reconfigurable modules (Section V). Section VI will show a case study and the paper is concluded in Section VII.

II. THE GOAHEAD TOOL AND DESIGN FLOW

Implementing reconfigurable systems requires a few more steps than the implementation of a static only system. The goal of GOAHEAD is to support and to automate these steps for enhancing design productivity and for avoiding an error prone design flow. The overall GOAHEAD philosophy is to automate low level design aspects that are required when implementing run-time reconfigurable systems. To achieve this, we abstract away from the target FPGA architecture (note that GOAHEAD permits system mitigation among different devices or even device families). The designer will mostly have a system level view where he has to think in modules, interfaces, geometrical aspects, or scheduling rather than in wires and switch matrix multiplexer settings. However, the tool provides several low level report and manipulation functions that are all accessible via a scripting interface and a comfortable GUI (see Figure 2). All complex commands that call other commands can generate a trace, such that all processes can be observed. Moreover, these traces can be put into a script file (optionally manipulated) and executed again. This low level scripting interface enables the building of very complex reconfigurable systems and can be used to integrate GOAHEAD with further projects (e.g., for improving the floorplanning, or implementing a component-based design methodology). The overall GOAHEAD design flow is illustrated in Figure 3 and the following sections explain the main steps.

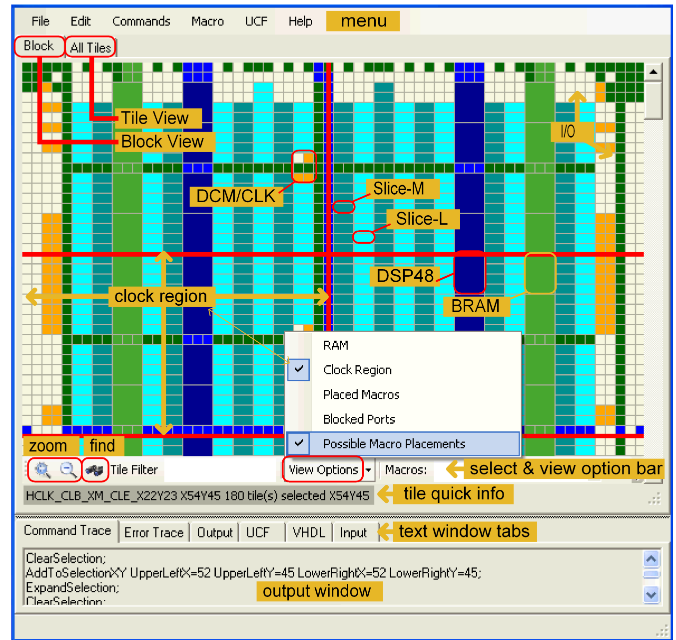

Figure 2. GOAHEAD GUI showing a Spartan-6 LX16 FPGAs.

A. Initial Planning

At the beginning of the design process, a planning step is required. Here, a design engineer has to define which parts of the system will be static (i.e., parts available at any time) and which parts (a set of modules) will be reconfigurable. In general, it is difficult to automatically decide which modules will be used mutually exclusively to each other at run-time. However, this process is in most cases straightforward to perform. Furthermore, it is also required to define partial modules. This can be done bottom-up by defining a partial region interface and by fitting modules into such a region. Alternatively, in a top-down design flow, partial modules are extracted from the hierarchy of a given design. Partial modules can have any design hierarchy and also generics are supported, but the modules must provide an entity containing only the partial module interface signals. This has to be done for each

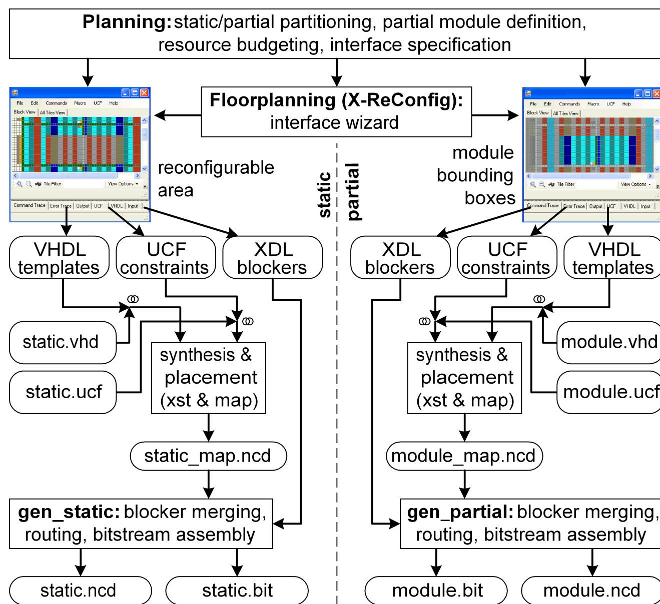

Figure 3. Simplified view on the different logic resource types (slices) provided on Spartan-6 FPGAs.

module separately. The union of all interfaces of modules that share a reconfigurable area defines the interface for the static part of the system. This interface includes all signals except the clocks, which are treated differently.

In addition, a resource budgeting is performed to determine the resource requirements (e.g., #LUTs, #BRAMs) for the static system and for the partial modules. For IP cores, the requirements are typically known. Otherwise, we can run an initial implementation (up to the technology mapping). The resulting netlist can then be analyzed by GOAHEAD. As the Xilinx vendor tools preserve hierarchy information, GOAHEAD can list for each hierarchy branch of the design the exact resource requirements. There is no restriction to perform the resource budgeting in one netlist or in separate netlists (for the different partial modules and the static system), because GOAHEAD can optionally filter hierarchy branches in a design.

B. Floorplanning

With the knowledge of the resource requirements of the partial modules, it has to be decided how many resources each reconfigurable area should provide. In the case of an island style reconfiguration where only one module will be hosted exclusively in an island, this is the maximum of each particular resource type (e.g., slices, BRAMs) over all modules sharing an island. In the case of recent Xilinx FPGAs, this value will be in most cases rounded up to the number of resources available in the height of a clock region (the smallest atomically reconfigurable unit on these devices). GOAHEAD can compute these values for a set of modules (given as individual netlists).

In the case that multiple modules should share a reconfigurable region at the same time, the resource requirements have to be manually specified. Here it must be considered that due to fragmentation it could be possible that the area must be larger than what is required with respect to the set of modules taking most resources at run-time.

The gathered information permits selection of the FPGA (capacity), definition of reconfigurable areas, and definition of module bounding boxes. GOAHEAD supports this via the GUI and the scripting interface. Area or bounding box definitions can be manually specified or automatically generated. The automatic mode follows the strategy to rise a reconfigurable area starting in the center of the device by following approximately the aspect ratio of the FPGA until the resource requirements are met. In this case, clock resource columns will be excluded and kept for the static system. This works reasonable well for single island style reconfiguration as the modules will use the same geometrical information.

In the initial planning phase, we defined module interfaces in terms of VHDL entities. GOAHEAD provides a wizard that can read module entities and build up the interface definition for a reconfigurable area. Such a definition contains signals with attributes such as input, output, or streaming (i.e. inputs and outputs on compatible interfaces on both sides of a module). In addition, the definition contains geometrical information of where to place the interface and a signal-to-wire assignment. This is very similar to the process of specifying a physical plug on a printed circuit board (PCB) where we also have to define positions and a pin assignment. The wire assignment can optionally include timing information for the signal propagation delays allowed on the static side and on the partial side to reach the interface (located at the border to a reconfigurable module). This information will be used to generate timing constraints for the Xilinx vendor tools. The GOAHEAD interface wizard can automatically create wire assignments. The wizard fits all single bit interface signals into vertically aligned interfaces that are placed at the left and/or right border of the reconfigurable area (or a module). The packing fits up to four signal wire connections in a single CLB and vector signals are grouped such that LSBs to MSBs are packed in a bottom up manner. This rule-based approach fits well to Xilinx FPGAs where the carry chain logic comprises four LUTs per CLB propagating in the same direction (see also Figure 6b))[1].

C. Static System Implementation

After performing the floorplaning, GOAHEAD creates all required VHDL templates and physical implementation constraints. The VHDL templates include communication macro instantiations (e.g., bus macros) and/or instantiations of connection primitives. From a system level perspective, this represents the logical plugs for the signals between the static system and the modules and this is very similar to instantiating an FPGA I/O pin primitive. For all macro instantiations, GOAHEAD creates placement constraints for the macros (derived from the interface definition). At the moment, the VHDL templates have to be included manually, while the UCF file

[1] In [13], [14], simulated annealing heuristics have been presented for the interface packing that could be straightforward integrated into the GOAHEAD framework. However, these approaches can take more than a day to process and optimize only the static system. Our rule-based approach considers more the partial modules where the routing is typically very congested.

manipulations can be performed automatically. However, an approach to automate the VHDL manipulation is presented in [13], and it is suitable for integration into our framework.

When implementing a reconfigurable system, we must strictly ensure that FPGA resources are used either for implementing the static system or for a partial module. For the primitives (e.g., Slices or BRAMs), Xilinx provides different possibilities and by default, GoAHEAD generates `CONFIG PROHIBIT` statements. For the static system, such a statement will be generated for each primitive located within a the reconfigurable region. Consequently, only the remaining primitives around the reconfigurable areas will be used for implementing the static system.

Unfortunately, Xilinx does not provide a comparable constraint for the routing resources. As an alternative, GoAHEAD creates blocker macros. These macros occupy a definable set of wires (e.g., all wires inside a reconfigurable area). Such a blocker is included into a design before starting the Xilinx vendor router. The router will recognize all blocked wires as already occupied and continue with the remaining routing resources. With the recent Xilinx design tools, every individual wire can be blocked in this way. Blocking has also been used by the ReCoBus-Builder [10] and in OpenPR [9], but GoAHEAD can create blockers more flexibly as shown in Section III and Section V.

As shown in Figure 3, the static part will be implemented completely independently from the partial module implementation. The modified VHDL code has to be compiled, and the result will be placed by following the extra constraints generated by GoAHEAD . This can be done using a third party synthesis tool or the Xilinx vendor tools (on the command line or in the ISE design environment). The final steps are carried out by a script (here `gen_static`). This script merges the blocker into the placed design using the Xilinx Design Language (XDL) [15]. The following routing is done by the Xilinx FPGA editor (the command line version). For the final routed design, the script runs a timing analysis and the bitstream generation. This is basically the same flow as described for OpenPR [9]. The main difference is that GoAHEAD uses blockers to provide clock signals in reconfigurable areas. This means that we basically include dummy primitives (e.g., slices or BRAMs) into the design and connect them to global clock nets and let the vendor tools create the final clock tree routing. In OpenPR, the tool adds the clock tree routing without using the vendor tools. Both approaches are completely transparent to the user and permit multiple clock domains.

D. Partial Module Implementation

The partial module implementation follows the same idea of constraining that was used for the static system. However, instead of generating prohibit statements and blockers *in* a region, these constraints will now be generated *around* it. Consequently, we will not use any primitive or routing outside the module bounding box. This is implemented by defining bounding boxes (according to the resource requirements) in GoAHEAD . By using the interface description that has been used for the static system, our tool generates VHDL templates, UCF constraints, and a blocker. These files will be used in

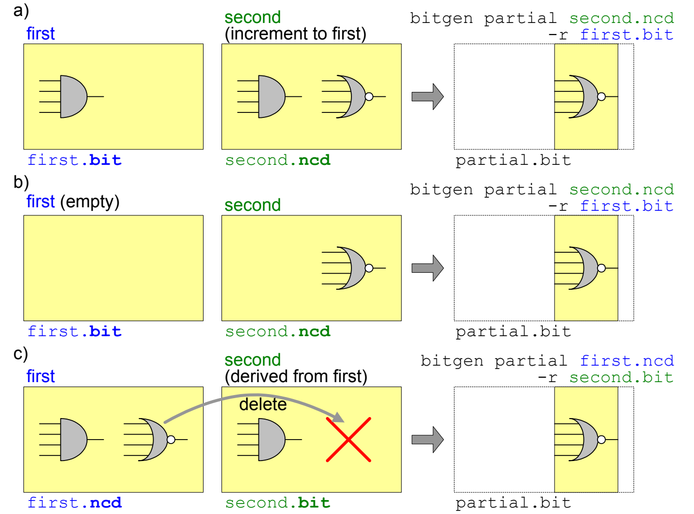

Figure 4. Partial bitstream generation using the differential bitstream option (`bitgen -r`). a) where the partial part is implemented as an increment to another (typically static) system. b) where the partial module is completely separated. c) as an alternative to b), we can delete the parts that should be added to the bitstream. Note, by changing first with second, a blanking bitstream can be generated.

a very similar way to that during the static system implementation. Note that the partial module implementation is completely decoupled from the static system implementation. Furthermore, all modules are implemented individually in the way described here. Consequently, there is no dependency between a partial module and the static system or any other partial module. The only shared part is the interface description.

As shown in Figure 3, another script (`gen_partial`) is used for the blocker insertion, final routing, and bitstream assembly. This script works similarly to the static routing script except for the final bitstream assembly, which has to create partial bitfiles. In previous versions of the Xilinx bitstream generation tools, there was an option for generating a partial bitstream for a definable region of the FPGA fabric. This option is unfortunately not available for recent FPGA families. However, there is an option to generate differential bitstreams using the `bitgen -r switch`. Without further options, the command is called as follows:

```
>bitgen to.ncd -r from.bit diff.bit
```

This command will create a partial bitstream that contains the differences to switch from the `from` file to the `to` file. The `to` file is always in the NCD netlist format and the `from` file is always a bitstream (`*.bit`). Consequently, before running this command, bitgen has to be called to generate the `from` file. By swapping the `from` file with the `to` file, we will generate a blanking bitstream that removes the partial module from the system (if loaded to the FPGA).

As illustrated in Figure 4, we support three different methods to generate partial bitstreams using `bitgen`:

a) *Incremental*: here, the partial module is generated as an increment to an existing design (typically a static system). This is the method used by the Xilinx vendor PR flows.
b) *Isolated*: this is the GoAHEAD default partial bitstream generation method. Here, the partial module is strictly

separated from any other parts of the system in order to generate a differential bitstream among a completely empty system. The isolation is supported by GOAHEAD with the command `CutOffFromDesign`.

c) *Remove*: here, the GOAHEAD `CutOffFromDesign` command is used to delete the part of the design for which a bitstream is going to be generated. By taking the original bitstream as the target (`to` file), a partial bitstream containing only the deleted parts is generated.

E. Further GOAHEAD Features

GOAHEAD is feature-rich and due to space limitations we cannot reveal the following features in more detail:

1) Hierarchical Reconfiguration: This is possible, because we can define a reconfigurable area inside a partial module bounding box in exactly the same way a reconfigurable area is defined in the static system. We used this to implement reconfigurable instruction set extensions in a partially reconfigurable softcore CPU.

2) Communication Architecture Synthesis: GOAHEAD can create homogeneous communication architectures for streaming ports that are required for slot or grid based reconfigurable systems. In such systems, a larger reconfigurable area is partitioned in regular tiles and the architecture provides interfaces in each tile such that all tiles provide the same footprint (see also Section IV-A). The streaming architecture is similar to existing approaches (e.g., [5], [10], [2]), but supports latest Xilinx FPGAs.

3) PR Simulation: We developed a library for cycle accurate RTL simulation of reconfigurable systems using multi island style reconfiguration. Here, the idea is to scan the ICAP port configuration data to determine which module is written where in order to (de)activate modules as they would be available at run-time. During reconfiguration, interface signals and partial module states will be forced to be undefined.

III. DIRECT WIRE BINDING

Many reconfigurable systems use bus macros to integrate reconfigurable modules into a system. In bus macros, one logic primitive is placed in the static system and another one in the reconfigurable area and wires between them are used to carry out the routing between the static and the partial part. By placing the macro on the partial area border, an interface signal to wire binding is achieved due to the internal bus macro routing that will be maintained through all implementation steps [8]. Interface signal wires work similarly to a physical plug located on a PCB and the binding of interface signals to wires has to be identical for the static system and all partial modules.

However, integrating partial modules using bus macros or proxy logic, has several drawbacks, including logic cost, additional latency, and restrictions on the module placement. GOAHEAD provides an alternative that circumvents these problems by binding the interface signals directly to the wires crossing the PR border without the help of logic resources. This is achieved by blocking. Here, it is important to understand that wire blocking basically specifies an *allocation* of FPGA routing resources that are used for a specific routing step, but

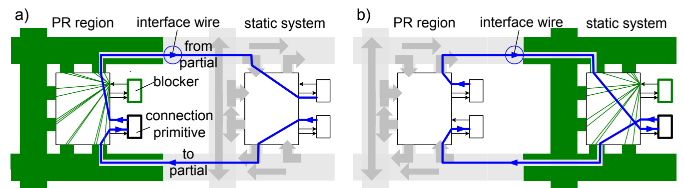

Figure 5. Direct wire binding. a) during the implementation of the static system, connection primitives act as a placeholders for the partial modules. A blocker placed *into* the reconfigurable region congests all wires except some tunnels for the interface wires. b) during the implementation of the partial modules, connection primitives act as a placeholder for the static system. A blocker placed *around* the reconfigurable region congests all wires except some tunnels for the interface signals. In both cases, the same wire was used for a specific signal and there will be no connection primitive remaining, if a partial module will be placed into the static system.

we cannot directly define the *binding*. In this case, binding means that we cannot define that a specific $signal_x$ has to be routed using a certain $wire_y$. The trick is to create blockers that leave only one option to route a path between the static system and a reconfigurable module.

As illustrated in Figure 5, this can be implemented very similarly to the original bus macro flow, by splitting the macro in two parts, one part for the static system and one for the modules. Note that the blockers are identical to the blockers that would have been created for a bus macro.

IV. MODULE RELOCATION AND COMPONENT-BASED DESIGN

The capability to isolate modules and to relocate them to different positions on an FPGA is essential for all more advanced reconfigurable systems (or for a component-based design methodology). For instance, this is required to be able to instantiate the same module multiple times in a system. Relocating modules requires that the FPGA provides the same kinds of resources with the same internal layout at different positions of the FPGA. This is commonly considered in the way that the layout of logic (CLB), memory (BRAM), or multiplier (DSP48) primitive columns has to fit a module at the target reconfigurable area on the FPGA fabric. However, this model does not reflect further aspects, including routing, timing and configuration bitstream aspects. We summarize all these aspects by the term *footprint* and a module footprint must match the footprint of the FPGA at the target position. The following sections will clarify this in more detail:

A. Resource Footprint

As already mentioned, the different primitive columns result in a heterogeneous layout which has to be considered when relocating modules. And in addition to CLB, BRAM, and DSP48 columns, the more recent Virtex-series FPGAs from Xilinx, consist of columns of I/O cells, clock generation blocks, and many further dedicated function blocks targeting domain specific applications (e.g., high-speed transceiver blocks, PCIe interface cores, or memory cores). The module resource footprint reflects the layout of primitives within a module which has to be provided at the target position by the FPGA.

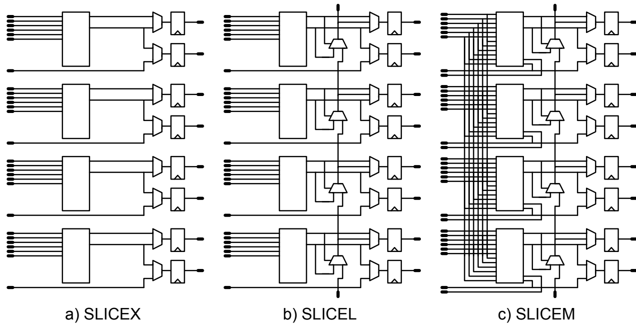

Figure 6. Simplified view of the different logic resource types (slices) provided on Spartan-6 FPGAs.

Furthermore, on Spartan-6 FPGAs, the logic primitives differ and there are a) simple LUT primitives, b) primitives with carry chain logic, and c) primitives with carry chain logic and distributed memory, as sketched in Figure 6. As a more complex primitive can work also as a simpler one, module relocation is possible, even of the logic resource footprint varies among different reconfigurable islands. For example, if a module uses only SLICEX logic primitives (logic only), it could also work at positions that provide SLICEL (with carry chain logic) or SLICEM (distributed memory) slice primitives.

B. Wire Footprint

Not only does the logic resource primitive footprint have to compatible, the same holds for the footprint of the routing resources. For example, on the smallest Spartan-3 FPGAs, there are no hexline wires available that exist on all larger FPGAs within the Spartan-3 family. This has to be considered when migrating modules among different FPGAs (e.g., for a component-based design methodology).

In general, it might be necessary that the static system occupies routing resources within a reconfigurable region. If a module demands those routing resources, module relocation is not possible due to a wire footprint mismatch.

C. Timing Footprint

There is also a timing footprint of the FPGA fabric that has to be considered when relocating modules. For example, the configuration logic on Xilinx FPGAs is arranged in a (hidden) column located in the center of the fabric. Consequently, this logic results at some points in longer routing delays. Furthermore, the clock signals on Xilinx FPGAs are routed in multiple horizontal paths from where they branch into vertical splines up and down. Relocating a module from an upwards clock spline to a downwards spline will consequently impact the timing. This is because a vertically routed signal within the module will propagate once with the clock wave and once against it.

When using a communication architecture to integrate reconfigurable modules in a flexible manner, this architecture will have an internal latency which may impact the attached partial module. The latency can be placement position dependent (e.g., if no registers are used at the interface of the communication architecture). Consequently, the timing footprint includes also the latency on interface signals.

D. Bitstream Footprint

On Xilinx FPGAs, modules can be relocated by changing an address field in the configuration bitstream header. However, on Xilinx Spartan-6 FPGAs, modules using only the LUT function generator functionality (i.e., not using carry chains or distributed memory) can be relocated on a netlist level (as mentioned in the resource footprint section). This is possible without any changes to the internal logic and routing of such a module. However, the bitstream encoding differs for CLB columns providing distributed memory (CLEXM CLB columns) as compared to other logic columns (see Figure 6 for an illustration of the different logic primitives). Consequently, it can require more than one partial module bitstream for supporting arbitrary module relocation.

E. Module Relocation in GOAHEAD

In order to ensure that a module will operate correctly at all intended positions, GOAHEAD can compose a netlist for any configuration state a system would have after loading a reconfigurable module onto an FPGA. For all these netlists, we run the timing analyzer and we can generate a corresponding partial bitstream. A timing violation will typically require a reimplementation of the module. For the bitstreams, GOAHEAD checks if more than the address field differs for all bitstreams generated for a specific module at the different placement position. If this is the case, more than one bitstream is required at run-time. This ensures a fit of the timing and bitstream footprint. Note that this process would be straightfoward to parallelize. Moreover, by decoupling modules at their interfaces (e.g. using registers at the module I/Os), we can prevent that different modules from interfering with each other. Consequently, each module can be verified individually and we do not have to create all possible permutations of module instances and placement positions.

To ensure a resource footprint and wire footprint match, GOAHEAD provides design rule checks and some powerful functions. The tool makes it possible to cut out arbitrary regions from a design in the granularity of a CLB or any other tile of the fabric. This cut-out can be stored as a module. As GOAHEAD has an internal wire database, it tracks if there are routing tracks crossing the border of the selection. In this case, the tool will add *module ports*. Such a port is defined by a name (derived from the net name), a direction (input or output) and the physical wire resource used to cross the module border. For a given system, GOAHEAD can scan for all possible module placement positions and the result can be shown in the GUI tile view or printed to the command line or a file. This scan checks module ports, wire violations, and resource mismatches.

As a counterpart to the cut-out function, GOAHEAD provides a module place function. To instantiate a module, a instance name and a placement position have to be specified. The instance name will be used as a prefix for all nets and primitives for this module when integrating it into another design (typically the static system). This permits multi module instantiations. In addition, the module ports are used to join nets. This includes a design rule check if ports of two adjacent parts (two modules or a module and the static system) match.

When following a reconfigurable design flow, this results in a fully routed netlist. This netlist could even be simulated (post place & route simulation). For a component based design methodology, it is alternatively possible to place multiple modules on the FPGA without respecting any ports. In this case, the Xilinx vendor router can be used to accomplish the final top level routing.

The functions provided for the component-based design part have commonalities with features provided by Torc [16] and RapidSmith [17] that use also XDL to perform netlist manipulations. However, GOAHEAD provides distinguished features for composing run-time reconfigurable systems and a component-based design flow can be arbitrarily combined with a partial design flow. When following the clock region constraint on Xilinx FPGAs, partial reconfiguration could then be used without side effects to other modules.

V. STATIC ROUTING IN RECONFIGURABLE AREAS

While the blocker approach presented in [9] permits module relocation, which is not available using the Xilinx vendor PR design tools, it has the drawback that no static routing can cross the reconfigurable region. In other words, a partial area in [9] is an *obstacle* for the static system routing and all signal paths have to route around this obstacle[2] which might result in considerable congestion and increased latency. Moreover, such static routing can be implemented in many cases "for free". The reason for this is that most partial modules will have an aspect ratio that is tall rather than wide. This is because of the column-wise reconfiguration scheme and because of the carry chain logic is arranged in vertical columns on Xilinx devices. In addition, there are routing wires spanning a wider distance and typically both sides of a wire (begin and end) have to be located inside the module bounding box. For example, on Spartan-6 FPGAs there exist quad lines that route four CLBs further. If we consider only horizontal quadlines, a partial module must be at least five CLB/BRAM/DSP48 columns wide to use quadlines at all. Otherwise, these quadlines would remain completely unused.

In GOAHEAD, which also uses blocking to constrain the routing, we permit static routing through reconfigurable areas as this was in particular required for highly utilized Spartan-6 FPGA designs. The fundamental principle behind partitioning the FPGA resources into one set allocated to the static system and another set of resources allocated to the reconfigurable modules is that both sets are separated from each other. As a consequence, we can allow static routing in a reconfigurable area as long as no module will demand the corresponding wire resources. The latter one can be ensured by blocking the specific wires used to cross a reconfigurable area during the physical implementation of the modules. In this case, the routing of the blocker will *speculatively* route all reserved paths through the reconfigurable region, regardless of whether they will be used or not by the static system. Consequently, the partial module bitstream does not have to be further adjusted at run-time.

[2]In Virtex FPGAs there are long lines available that can hop over a reconfigurable area. However, there are only a few long lines available and these wires are not available at all on Spartan-6 FPGAs.

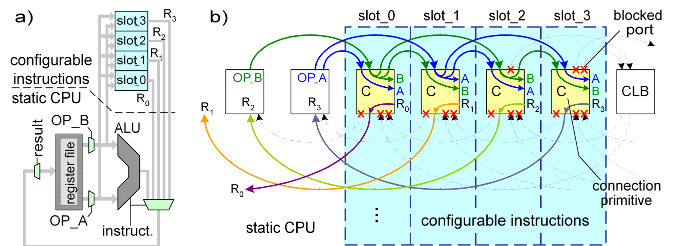

Figure 7. Reconfigurable instruction set extension. a) RTL b) FPGA implementation showing the direct wire communication architecture.

Using this approach has some limitations. While it is in most cases noncritical to reserve plenty of horizontal wires for the static routing, vertical wires are typically heavily used by the partial modules themselves [18]. Note that there are up to 256 horizontal quadlines available per direction within the height of a clock region (16 CLBs in height) on a Spartan-6 FPGA. In addition, we can only define a wire allocation, but not an exact routing by blocking, as discussed in Section III. This has two consequences: 1) the allocated wires for the static routing should not leave any freedom to the router; and 2) the allocated wires should not interfere with the communication architecture used to integrate partial modules. The latter constraint is bound to the direct wire integration technique (Section III) that is also based on a neat wire allocation and there are no aftereffects when using bus macros. The first constraint removes the need to manipulate a partial module bitstream in the case a reconfigurable module should work at different reconfigurable areas on the FPGA (that are assumed to have a different static routing).

GOAHEAD can automatically generate all constraints for the blockers that are required to allocate a definable number of wires to be used as static routing within a reconfigurable area. This is supported for a straight crossing in any horizontal and/or vertical direction. By applying this feature after the communication architecture definition, no wires will be allocated that might interfere with the communication architecture. This is possible as GOAHEAD has full knowledge about all possible switch matrix settings.

VI. CASE STUDY

GOAHEAD can implement complex reconfigurable systems and one particularly challenging example is the efficient integration of reconfigurable custom instructions into a softcore CPU. The problem is difficult, because there are many signals to be connected within a relatively small area. In this example, we will extend a 32-bit MIPS softcore running on a Spartan-6 FPGA with up to four different instructions that are freely placeable into a reconfigurable area that is tiled into four slots. As shown in Figure 7a), this requires the connection of at least three 32-bit signal vectors per resource slot. We chose a slot to be one CLB wide and eight CLBs in height. This corresponds to two 6-input LUTs per result bit. Note that some modules could perform valuable work without using the LUTs at all. For instance, a bit permutation function would be basically only wiring but would easily save more than a hundred MIPS instructions. However, larger instructions can simply allocate multiple consecutive slots.

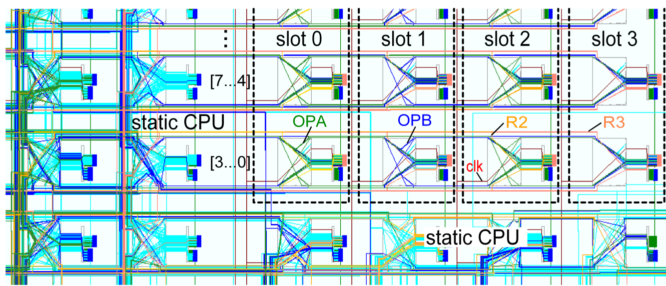

Figure 8. Regular communication architecture implementation for integrating custom instructions into a MIPS CPU.

Figure 7b) shows the communication architecture that neatly uses the different wire resources available on Spartan-6 FPGAs to implement direct wire binding, as introduced in Section III. The connection primitives in the figure are placeholders for the reconfigurable modules and will be replaced by reconfigurable modules (custom instructions) at run-time. Due to space limitations we only show the static implementation of the CPU. It can be seen from Figure 7b) that the communication architecture is quite complex. However, the corresponding GOAHEAD script can be easily reused for different systems like a black box.

An FPGA editor screen-shot of the reconfigurable system is shown in Figure 8 We have been able to connect 2 times 4 input signals in each CLB of the reconfigurable area (for the two operands) and 4 wires are connected individually back to the static system from each CLB. The whole communication architecture is implemented using only the FPGA routing fabric and no further logic overhead will affect the run-time system. For comparison, implementing the same system using bus macros would require for four slots $4 \cdot 3 \cdot 32 \cdot 2 \cdot \frac{1}{4} = 196$ slices (assuming the connection of four wire ports per slice). However, the whole reconfigurable region is in our example just 64 slices large.

In this example, module bitstreams are not relocatable as we don't use the full clock region height as a reconfigurable region. The number of instructions than can be integrated into such a system is virtually unbounded and we implemented several custom instructions (e.g., CRC checksum, permute functions, or a 64 bit parity checker) that all save more than one hundred MIPS instructions. Note that custom instructions could in principle be provided as part of a binary program.

VII. CONCLUSIONS

In this paper we introduced our new PR tool GOAHEAD. The tool, documentation, and ready-to-start design examples are available on [12]. GOAHEAD provides unique features that are required for implementing run-time reconfigurable systems on latest Xilinx FPGAs, such as Spartan-6 FPGAs. This includes static routing in reconfigurable modules, module relocation, resource aware bitstream generation, and integration of partial modules without logic overhead. Moreover, the tool is designed for usability and it abstracts most of the low level details of the FPGA fabric from the design engineer.

An addition to the development of more examples on more FPGA boards, we are currently investigating how GOAHEAD can be used to parallelize the place and route implementation phase on a compute farm.

ACKNOWLEDGMENT

This work is supported by the Norwegian Research Council under grant 191156V30.

REFERENCES

[1] S. N. und Anantha Chandrakasan, *Leakage in Nanometer CMOS Technologies*. Springer, 2006.

[2] H. Kalte and M. Porrmann, "Context Saving and Restoring for Multitasking in Reconfigurable Systems," in *Proceedings of the 15th International Conference on Field Programmable Logic and Applications (FPL)*, aug 2005, pp. 223–228.

[3] P. Athanas, J. Bowen, T. Dunham, C. Patterson, J. Rice, M. Shelburne, J. Suris, M. Bucciero, and J. Graf, "Wires On Demand: Run-Time Communication Synthesis for Reconfigurable Computing," in *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, August 2007, pp. 513–516.

[4] G. Brebner, "The Swappable Logic Unit: A Paradigm for Virtual Hardware," in *IEEE Symposium on FPGAs for Custom Computing Machines (FPGA)*, K. L. Pocek and J. Arnold, Eds., Los Alamitos, CA, April 1997, pp. 77–86.

[5] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular Dynamic Reconfiguration in Virtex FPGAs," *IEE*, vol. 153, no. 3, pp. 157–164, 2006.

[6] Xilinx Inc., "Partial Reconfiguration User Guide (Rel 13.2)," 2011, available online: `www.xilinx.com/support /documentation/sw_manuals/xilinx13_2/ug702.pdf`.

[7] Mark Bourgeault, "Alteras Partial Reconfiguration Flow," 2011, available online: `http://www.eecg.utoronto.ca/˜jayar /FPGAseminar/FPGA_Bourgeault_June23_2011.pdf`.

[8] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Invited Paper: Enhanced Architecture, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs," in *Proceedings of the 16th International Conference on Field Programmable Logic and Application (FPL)*, Aug 2006, pp. 1–6.

[9] A. A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood, "Openpr: An open-source partial-reconfiguration toolkit for xilinx fpgas," in *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*. IEEE Computer Society, 2011, pp. 228–235.

[10] D. Koch, C. Beckhoff, and J. Teich, "ReCoBus-Builder– a Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs," in *Proceedings of International Conference on Field-Programmable Logic and Applications (FPL 08)*, Heidelberg, Germany, Sep. 2008, pp. 119–124.

[11] S. Koh and O. Diessel, "COMMA: A Communications Methodology for Dynamic Module Reconfiguration in FPGAs," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Los Alamitos, CA, USA: IEEE Computer Society, 2006, pp. 273–274.

[12] CosReCos project website: `http://www.mn.uio.no/ifi/english/research/projects/cosrecos/`.

[13] S. Yousuf and A. Gordon-Ross, "DAPR: Design Automation for Partially Reconfigurable FPGAs," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. CSREA Press, June 2010.

[14] J. M. Carver, R. N. Pittman, and A. Forin, "Automatic Bus Macro Placement for Partially Reconfigurable FPGA Designs," in *Proceeding of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. New York, NY, USA: ACM, 2009, pp. 269–272.

[15] C. Beckhoff, D. Koch, and J. Torresen, "The Xilinx Design Language (XDL): Tutorial and Use Cases," in *Proceedings of the 6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, Montpellier, France, 2011, pp. 1–8.

[16] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, "Torc: towards an open-source tool flow," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA)*. ACM, 2011, pp. 41–44.

[17] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. E. Nelson, and B. Hutchings, "RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs," in *Proceedings of Int. Conf. on Field-Programmable Logic and Applications (FPL)*. IEEE, 2011, pp. 349–355.

[18] D. Koch, C. Beckhoff, and J. Teich, "A Communication Architecture for Complex Runtime Reconfigurable Systems and its Implementation on Spartan-3 FPGAs," in *Proceedings of the 17th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. Monterey, California, USA: ACM, Feb. 2009, pp. 233–236.